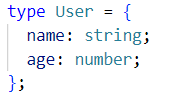
📘 Chapter 7: Type vs Interface

Both type and interface allow you to define the shape of an object in TypeScript.

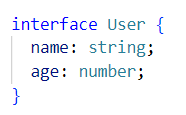
They might look the same, but they function very differently beneath the surface.

**Basic Syntax:**

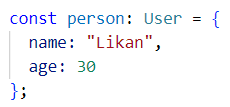
**Type -**



**Interface -**



**Usage in object -**



***Similarities –***

✅ Both can describe object shapes  
✅ Both support optional properties  
✅ Both support readonly properties  
✅ Both support function types  
✅ Both support extending other types/interfaces

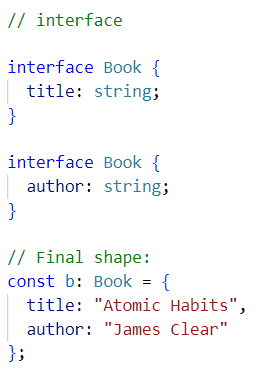
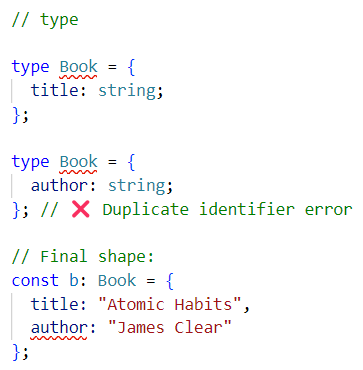
***Differences (with examples) -***

**1. Re-opening (Declaration Merging) -**

interface allows you to define it in multiple places - and it merges automatically

type cannot be re-opened once declared

Example:

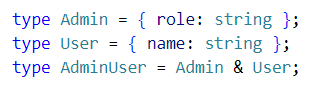


**2. Union & Intersection -**

type supports unions.

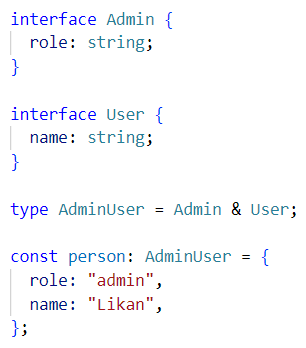


type supports intersections.



Interface Supports Intersections.

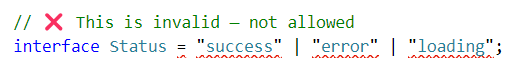
You can intersect interfaces just like types using the **&** operator.



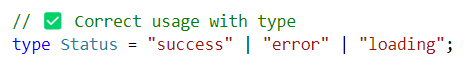
Even though Admin and User are defined using interface, the intersection (Admin & User) is defined using **type**, which is totally valid.

### interface does **not** support union types directly.

You **cannot** define a union like this using interface:



That’s **only allowed with** type:



### Why?

* interface is designed to define the **structure of an object**.
* types can describe not only objects, but also primitives, unions, intersections, functions, tuples, and more.

## **3. Computed Properties + Conditional Types** **(Advanced Meta-Programming)**

### Only type (not interface) can do this kind of advanced logic.

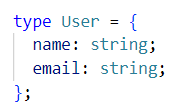
### *Part 1:* ***Computed Properties with Mapped Types -***

### 

### What’s happening here?

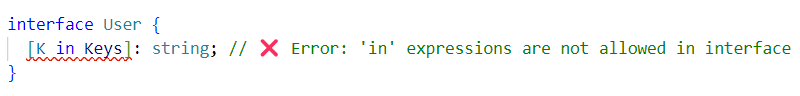
* Keys is a union type: "name" | "email"
* [K in Keys] means: "for each K in Keys"

User becomes -



This is **meta-programming** - generating types **dynamically** using logic.

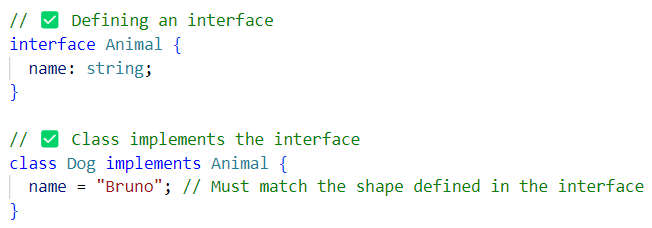
### 🚫 interface can't do this -



## **4. Implementing & Extending in Classes**

### Both interface and type can be used with class via implements.

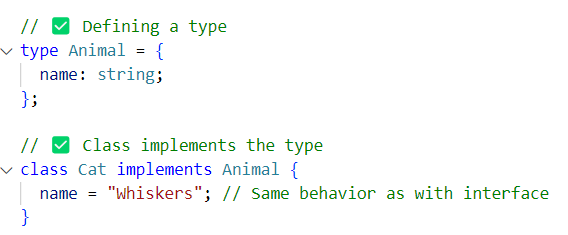
### Example 1: Using interface with class



**🧠 What happens here?**

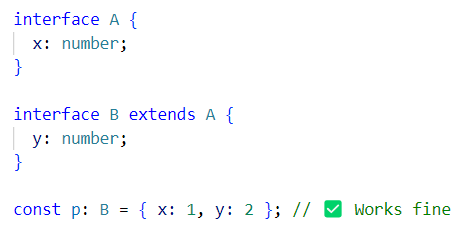
* The class Dog is **guaranteed** to have a name property of type string.
* TypeScript will show an error if name is missing or the type is wrong.

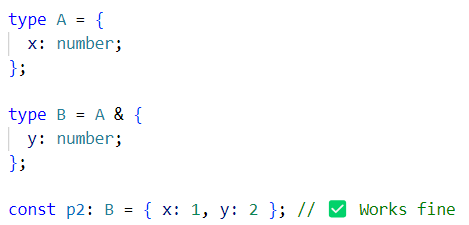
### Example 2: Using type with class



🧠 You get **the same result** as using interface.

Example 3 -





Same result - just different approach.

**🛠 Use interface when:**

✅ You’re describing **objects** (like user, product, props)  
✅ That object might **grow / extend later**  
✅ You want to **reuse and extend** it easily  
✅ You're making **React props or class shapes**

Think: **Interfaces = best for objects and React props**

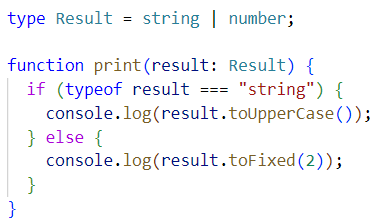
**⚡ Use type when:**

✅ You need **union types** — A | B  
✅ You need **intersection types** — A & B  
✅ You want **advanced / fancy types** (conditional, utility, mapped)  
✅ You want to build **complex logic-based types**

Think: **Types = more flexible for advanced type features**

Regular Union Vs Discriminated Union -

### 1. Regular Union (❌ No discriminant key)

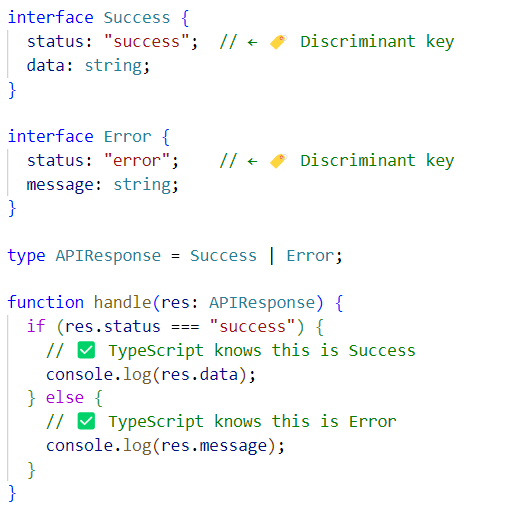


Here you rely on **typeof** or **manual checks** to **narrow** the type.

### 2. Discriminated Union or 🏷️ Tagged Union

### What is a **Tagged Union** (Discriminated Union)?

A **Tagged Union** is a **union of types** where **each type has a unique "tag" property** (usually a string literal like "type", "kind", or "status"). This tag helps TypeScript **automatically figure out** which specific type you're working with during **type narrowing**.



In this example status is the **discriminant key** (aka the **tag**). APIResponse is the union type.

This is called a **Discriminated Union** because each type in the union is "tagged" with a unique status key. Based on the value of status, we can **narrow down the type**.

**Type narrowing** helps TypeScript understand the **exact or specific type at compile time,** enabling safer and more accurate code.

A discriminated union:

* Requires a common literal field (status, type, etc.).
* Allows type-safe checks without manual type assertions.
* Keeps code clean, safe, and readable.
* Works best when combining multiple object shapes.

Interview Questions

***Q1: Can you use a type and interface interchangeably?***

Sometimes yes (for object shapes), but only type supports unions/conditions.

***Q2: Can interfaces merge? What about types?***

Only interfaces merge automatically.

***Q3: When would you choose type over interface?***

When using advanced features like unions, mapped types, or conditional types choose types.

***Q4: Which one is preferred in public libraries (like React)?***

interface - because interface can be extended later by users, so it's easier to add more properties without changing the original code.